Functional Programming

Tim Sheard, March 2015

Categories, Algebraic and CoAlgebraic Programs

- Categories
- Functors
- F Algebras
- Initial and Final Algebras
- Induction and CoInduction

Lecture 15 — Tim Sheard — Tim Sheard

Categories

A category is a mathematical structure

A set of objects

A set of arrows between objects

Every object, A, has an arrow to itself id_A

Arrows are transitive (composable)

 $F: A \rightarrow B$

 $G: B \rightarrow C$

Then there exists $(G \circ F) : A \rightarrow C$

We sometimes write this as $(F ; G) : A \rightarrow C$

Functor

A Functor is a mapping between categories

The mapping has two parts
One part maps objects
The other part maps arrows

If T is a functor between categories C and D, then T is used in two different ways.

If X is an object in C, then T X is an object in D

If f is an arrow in C between objects A and B, then T f is an arrow in D between T A and T B

_ecture 15 \longrightarrow Tim Sheard \longrightarrow 3

Functor Laws

Let A be an object in C Then id_A is a function in C Then $T id_A$ is a function in D, in fact it must be $id_{(T A)}$

Let g: P -> Q and f: Q -> R in C

Then (f ° g) is a function in C between P and R

Then T (f ° g) is a function in D between T P and T R, in fact it must be the function

T f ° T g in D

Functors in Haskell

```
class functor t where
  fmap:: (a -> b) -> t a -> t b
```

Note "t" is the mapping on objects, and "fmap" is the mapping on arrows.

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

_ecture 15 ———— Tim Sheard ———— 5

Endo Functors

An endo-functor is a functor from a category to itself.

```
In the Haskell world, there is a category (call it H) where the objects are types (like Int, Bool, [Int] etc), and the arrows are functions (like (+1), length, reverse)
```

Functors in Haskell are endofunctors from H to H

Haskell functors

Just about any first order datatype with one parameter in Haskell defines a functor

```
data L x = Nil | Cons Int x
data T x = Tip | Fork x Int x
data E x =
   Const Int | Add x x | Mult x x
```

Why must the data structure be first order (i.e. without embedded functions)?

Algebras and Functors

An F-algebra over a carrier sort x is set of functions (and constants) that consume an F x object to produce another x object.

In Haskell we can simulate this by a data definition for a functor (F x) and a function (F x) -> x

```
data Algebra f c = Algebra (f c -> c)
data F1 x = Zero | One | Plus x x
data ListF a x = Nil | Cons a x
```

Note how the constructors of the functor play the roles of the constants and functions.

Lecture 15 — Tim Sheard — Tim Sheard

Examples

```
f :: F1 Int -> Int
f Zero = 0
f One = 1
f (Plus x y) = x+y
g :: F1 [Int] -> [Int]
g Zero = []
g One = [1]
g (Plus x y) = x ++ y
alg1 :: Algebra F1 Int
alg1 = Algebra f
alg2 :: Algebra F1 [Int]
alg2 = Algebra g
```

= Tim Sheard ————

More Examples

```
data ListF a x = Nil | Cons a x
```

```
h :: ListF b Int -> Int
h Nil = 0
h (Cons x xs) = 1 + xs
```

```
alg3 :: Algebra (ListF a) Int
alg3 = Algebra h
```

Initial Algebra

An initial Algebra is the set of terms we can obtain be iteratively applying the functions to the constants and other function applications.

This set can be simulated in Haskell by the data definition:

```
data Initial alg = Init (alg (Initial alg))
```

Here the function is:

```
Init :: alg (Init alg) -> Init alg
```

 $f :: T \times -> x$

Note how this fits the (T x -> x) pattern.

Example elements of Initial Algebras

```
ex1 :: Initial F1
ex1 = Init(Plus (Init One) (Init Zero))

ex2 :: Initial (ListF Int)
ex2 = Init(Cons 2 (Init Nil))

initialAlg :: Algebra f (Initial f)
initialAlg = Algebra Init
```

Defining Functions

We can write functions by a case analysis over the functions and constants that generate the initial algebra

F-algebra homomorphism

An F-algebra, f, is said to be initial to any other algebra, g, if there is a UNIQUE homomorphism, from f to g (this is an arrow in the category of F-algebras).

We can show the existence of this homomorphism by building it as a datatype in Haskell.

Note: that for each "f", (Arrow f a b) denotes an arrow in the category of f-algebras.

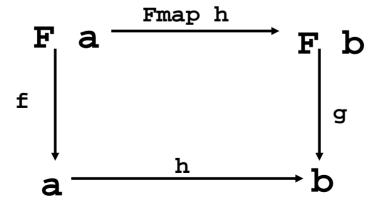
```
data Arrow f a b =
    Arr (Algebra f a) (Algebra f b) (a->b)
-- plus laws about the function (a->b)
```

F-homomorphism laws

For every Arrow

(Arr (Algebra f) (Algebra g) h)

it must be the case that



Tim Sheard 1

Lecture 15

Existence of h

To show the existence of "h" for any F-Algebra means we can compute a function with the type (a -> b) from the algebra. To do this we first define cata:

```
cata :: Functor f => (Algebra f b) -> Initial f -> b

cata (Algebra phi) (Init x) =
    phi(fmap (cata (Algebra phi)) x)

exhibit :: Functor f =>
    Algebra f a -> Arrow f (Initial f) a

exhibit x = Arr initialAlg x (cata x)

    Initial
    Algebra
    Algebra
    Algebra
    Algebra
    Algebra
```

16

Tim Sheard

Writing functions as cata's

Lots of functions can be written directly as cata's

```
len2 x = cata (Algebra phi) x
   where phi Nil = 0
      phi (Cons x n) = 1 + n

app2 x y = cata (Algebra phi) x
   where phi Nil = y
      phi (Cons x xs) = Init(Cons x xs)
```

Induction Principle

With initiality comes the inductive proof method. So to prove something (prop x) where x::Initial A we proceed as follows

```
prop1 :: Initial (ListF Int) -> Bool
prop1 x =
  len(Init(Cons 1 x)) == 1 + len x
```

Prove: prop1 (Init Nil)

Assume prop1 xs

Then prove: prop1 (Init (Cons x xs))

Induction Proof Rules

```
For an arbitrary F-Algebra, we need a function from
```

```
F(Proof prop x) -> Proof prop x
```

CoAlgebras

An F-CoAlgebra over a carrier sort x is set of functions (and constants) whose types consume x to produce an F-structure

```
data CoAlgebra f c = CoAlgebra (c -> f c)
unCoAlgebra (CoAlgebra x) = x

countdown :: CoAlgebra (ListF Int) Int
countdown = CoAlgebra f
  where f 0 = Nil
    f n = Cons n (n-1)
```

_ecture 15 $-\!-\!-\!-$ Tim Sheard $-\!-\!-\!-$ 20

Stream CoAlgebra

The classic CoAlgebra is the infinite stream

data StreamF n x = C n x

Note that if we iterate StreamF, there is No nil object, all streams are infinite. What we get is an infinite set of observations (the n-objects in this case).

Lecture 15 $-\!-\!-\!-\!-$ Tim Sheard $-\!-\!-\!-\!-$ 21

Examples

We can write CoAlgebras by expanding a "seed" into an F structure filled with new seeds.

```
seed -> F seed
```

The non-parameterized slots can be filled with things computed from the seed. These are sometimes called observations.

```
endsIn0s ::
   CoAlgebra (StreamF Integer) [Integer]
endsIn0s = CoAlgebra f
   where f [] = C 0 []
    f (x:xs) = C x xs
```

More Examples

```
split :: CoAlgebra F1 Integer
split = CoAlgebra f
 where f 0 = Zero
        f 1 = One
        f n = Plus (n-1) (n-2)
fibs :: CoAlgebra (StreamF Int) (Int,Int)
fibs = CoAlgebra f
 where f (x,y) = C(x+y)(y,x+y)
```

Final CoAlgebras

Final CoAlgebras are sequences (branching trees?) of observations of the internal state. This allows us to iterate all the possible observations. Sometimes these are infinite structures.

```
data Final f = Final (f (Final f))
unFinal :: Final a -> a (Final a)
unFinal (Final x) = x

finalCoalg :: CoAlgebra a (Final a)
finalCoalg = CoAlgebra unFinal
```

Lecture 15 $-\!-\!-\!-\!-$ Tim Sheard $-\!-\!-\!-\!-$ 24

Example Final CoAlgebra elements

```
f1 :: Final (ListF a)
f1 = Final Nil

ones :: Final (StreamF Integer)
ones = Final(C 1 ones)
```

Iterating

We can write functions producing elements in the sort of Final CoAlgebras by expanding a "seed" into an F structure filled with observations and recursive calls in the "slots". Note then, that all thats really left is the observations.

```
nats :: Final (StreamF Integer)
nats = g 0
where g n = Final (C n (g (n+1)))
```

_ecture 15 $-\!-\!-\!-$ Tim Sheard $-\!-\!-\!-$ 26

More Examples

```
data NatF x = Z \mid S x
omega :: Final NatF
omega = f undefined
  where f x = Final(S(f x))
n :: Int -> Final NatF
n x = f x
  where f 0 = Final Z
        f n = Final(S (f (n-1)))
```

Lecture 15 $-\!-\!-\!-\!-$ Tim Sheard $-\!-\!-\!-\!-$ 27

CoHomomorphisms

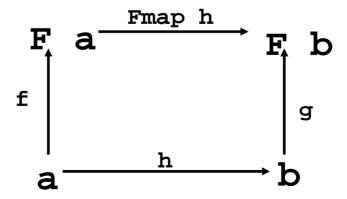
A CoHomomorphism is an arrow in the category of F-CoAlgebras

```
data CoHom f a b =
   CoHom (CoAlgebra f a) (CoAlgebra f b) (a->b)

For every arrow in the category
   (CoHom (CoAlgebra f) (CoAlgebra g) h)

it must be the case that

covalid :: (Eq (f b), Functor f) => CoHom f a b -> a -> Bool
   covalid (CoHom (CoAlgebra f) (CoAlgebra g) h) x = fmap h (f x) == g(h x)
```



Final CoAlegbra

A F-CoAlgebra, g, is Final if for any other F-CoAlgebra, f, there is a unique F-CoAlgebra homomorphism, h, from f to g.

We can show its existence be building a function that computes it from the CoAlgebra, f.

```
ana :: Functor f =>
          (CoAlgebra f seed) -> seed -> (Final f)
ana (CoAlgebra phi) seed =
  Final(fmap (ana (CoAlgebra phi)) (phi seed))
exhibit2 :: Functor f =>
     CoAlgebra f seed -> CoHom f seed (Final f)
exhibit2 x = CoHom finalCoalg x (ana x)
```

Examples

We use and to iteratively unfold any coAgebra to record its observations

```
final1 = ana endsIn0s
final2 = ana split
final3 = ana fibs
```

```
endsIn0s = CoAlgebra f
  where f [] = C 0 []
     f (x:xs) = C x xs

split = CoAlgebra f
  where f 0 = Zero
     f 1 = One
     f n = Plus (n-1) (n-2)

fibs :: CoAlgebra (StreamF Int) (Int,Int)
fibs = CoAlgebra f
  where f (x,y) = C (x+y) (y,x+y)
```

```
tak :: Num a => a -> Final (StreamF b) -> [b]
tak 0 _ = []
tak n (Final (C x xs)) = x : tak (n-1) xs
fibs5 = tak 5 (final3 (1,1))
```

= Lecture 15 \longrightarrow Tim Sheard \longrightarrow 30

CoAlgebras and ObjectOrientation

Lets use CoAlgebras to represent Points in the 2-D plane as we would in an OO-language

```
data P x = P { xcoord :: Float
              , ycoord :: Float
              , move :: Float -> Float -> x}
pointF :: (Float,Float) -> P (Float,Float)
pointF (x,y) = P \{ x coord = x \}
                   , ycoord = y
                   , move = \ \ m \ n \rightarrow (m+x,n+y) 
type Point = CoAlgebra P (Float,Float)
point1 :: Point
point1 = CoAlgebra pointF
```

Lecture 15 \longrightarrow Tim Sheard \longrightarrow 3°